Composing Distributed Systems: Overcoming the Interoperability Challenge

Valérie Issarny, Amel Bennaceur

Composing Distributed Systems: Overcoming the Interoperability Challenge

Valérie Issarny and Amel Bennaceur

Inria Paris-Rocquencourt, France
{firstname.lastname}@inria.fr

Abstract. Software systems are increasingly composed of independently-developed components, which are often systems by their own. This composition is possible only if the components are *interoperable*, i.e., are able to work together in order to achieve some user task(s). However, interoperability is often hampered by the differences in the data types, communication protocols, and middleware technologies used by the components involved. In order to enable components to interoperate despite these differences, mediators that perform the necessary data translations and coordinate the components' behaviours appropriately, have been introduced. Still, interoperability remains a critical challenge for today's and even more tomorrow's distributed systems that are highly heterogeneous and dynamic. This chapter introduces the fundamental principles and solutions underlaying interoperability in software systems with a special focus on protocols. First, we take a software architecture perspective and present the fundamentals for reasoning about interoperability and bring out mediators as a key solution to achieve protocol interoperability. Then, we review the solutions proposed for the implementation, synthesis, and dynamic deployment of mediators. We show how these solutions still fall short in automatically solving the interoperability problem in the context of systems of systems. This leads us to present the solution elaborated in the context of the European CONNECT project, which revolves around the notion of emergent middleware, whereby mediators are synthesised on the fly. We consider the GMES (Global Monitoring of Environment and Security) initiative and use it to illustrate the different solutions presented.

Keywords: Architectural mismatches, Interoperability, Mediator synthesis, Middleware.

1 Introduction

Modern software systems are increasingly composed of many components, which are distributed across the network and collaborate to perform a particular task. These components, often being complex systems themselves, led to the emergence of what is known as "systems of systems" [32]. The realisation of a system of systems depends on the ability to achieve *interoperability* between its different component systems. Traditionally, *"Interoperability characterises the extent by*

which two implementations of systems or components from different manufacturers can co-exist and work together by merely relying on each other's services as specified by a common standard" [48]. However, assuming the reliance on a common standard is often unrealistic given that components are increasingly highly heterogeneous (from the very tiny thing to the very large cloud) and developed (from design to deployment) independently without knowing the systems with which they will be composed. As a result, even though the services that component systems (or *components* for short) require or provide to each other are compatible at some high-level of abstraction, their implementations may encompass many differences that prevent them from working together. Hence, we re-state the definition of interoperability to *components that require and provide compatible high-level functionalities and can be made to work together*. We qualify such components as being *functionally compatible*.

In order to make functionally-compatible components work together, we must reconcile the differences between their implementations. The differences may be related to the data types or the format in which the data are encapsulated, that is, *data heterogeneity*. Differences may also concern the protocols according to which the components interact, that is, *behavioural heterogeneity*, which is the main focus of this chapter. Middleware, which is a software logically placed between the higher layer consisting of users and applications, and the layer underneath consisting of operating systems and basic communication facilities, provides an abstraction that facilitates the development of applications despite the heterogeneity of the underlying infrastructure. However, the abstractions defined by the middleware constrain the structure of the data that components exchange and the coordination paradigm according to which they communicate. This makes it impossible for components implemented using different middleware technologies to interoperate. As a result, interoperability must be considered at both application and middleware layers. To achieve interoperability between components featuring data and behavioural heterogeneity, intermediary software entities, called *mediators*, are used to perform the necessary translations of the data exchanged and to coordinate the components' protocols appropriately [52]. Using mediators to achieve interoperability has received a great deal of interest and led to the definition of a multitude of solutions, both theoretical and practical, for the specification, synthesis, and deployment of mediators, although they are predominantly oriented toward design time.

With the growing emphasis on spontaneous interaction whereby components are discovered at runtime and need to be composed dynamically, mediators can no longer be specified or implemented at design time. Rather, they have to be synthesised and deployed on the fly. Therefore, the knowledge necessary for the synthesis of mediators must be represented in a form that allows its automated processing. Research on knowledge representation in general, and ontologies in particular, has now made it possible to model and automatically reason about domain information crisply, if not with the same nuanced interpretation that a developer might [45]. Semantic Web Services are an example of the use of ontologies in enabling mediation on the fly [38]. However, they are restricted to

interoperability at the application layer, assuming that the composed components are implemented using the same middleware.

Acknowledging the extensive work on fostering interoperability, while at the same time recognising the increasing challenge that it poses to developers, this chapter provides a comprehensive review of the interoperability challenge, from its formal foundations to its automated support through the synthesis of mediators. The work that is reported extensively builds on the result of the European collaborative project CONNECT[1], which introduced the concept of *emergent middleware* and related enablers so as to sustain interoperability in the increasingly connected digital world. An emergent middleware is a dynamically generated distributed system infrastructure for the current operating environment and context, which allows functionally-compatible systems to interoperate seamlessly.

This chapter is organised as follows. In Section 2, we introduce the GMES case study, which we use throughout the chapter to illustrate the different solutions to interoperability. In Section 3, we take a software architecture perspective to understand and further formalise the interoperability problem in the case of systems of systems, which are characterised by the extreme heterogeneity of their components and the high-degree of dynamism of the operating environment. In Section 4, we survey the approaches to achieving interoperability from (i) a *middleware perspective* where we concentrate on the effort associated with the implementation of mediators, (ii) a *protocol perspective* where we are concerned with the synthesis of mediators based on the behavioural specification of component systems, thereby greatly facilitating the developer's task and further promoting software correctness, and (iii) a Semantic Web perspective where we focus on the fully automated synthesis of mediators at runtime, so as to enable on-the-fly composition of component systems in the increasingly open and dynamic networking environment. Following this state of the art review, in Section 5, we introduce a multifaceted approach to interoperability which brings together the different perspectives in order to provide a solution to interoperability based on the automated synthesis of mediators and their dynamic deployment, which we call emergent middleware. Finally, in Section 6, we conclude on where interoperability stands in today's systems and present directions for future work.

2 GMES: A System of Systems Case Study

To highlight the interoperability challenge in systems of systems, we consider one representative application domain, that of global monitoring of the natural environment, as illustrated by the GMES[2] initiative. GMES is the European Programme for the establishment of a European capacity for Earth Observation. A special interest is given to the support of emergency situations across different European countries [22]. In emergency situations, the context is highly dynamic and involves highly heterogeneous components that interact in order to perform the different tasks necessary for decision making. The tasks include,

[1] `http://www.connect-forever.eu/`
[2] Global Monitoring for Environment and Security – `http://www.gmes.info/`

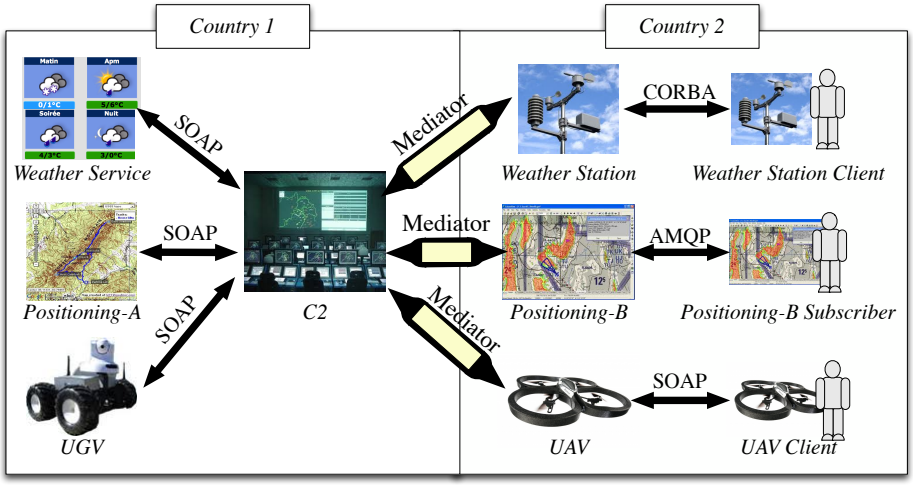

Fig. 1. The GMES use case

among others, collecting weather information, and capturing video using different devices. GMES makes a strong case of the need for on-the-fly solutions to interoperability in systems of systems. Indeed, each country defines an emergency management system that composes different components, which interact according to standards proper to the country. However, in special circumstances, assistance may come from other countries, which bring their own components defined using different standards.

Figure 1 depicts the case where the emergency system of *Country 1* is composed of a Command and Control centre (*C2*) which takes the necessary decisions for managing the crisis based on the information about the weather provided by the *Weather Service* component, the positions of the various agents in field given by *Positioning-A*, and the video of the operating environment captured by *UGV* (Unmanned Ground Vehicle) robots with sensing capabilities. The different components use SOAP[3] to communicate. *Country 2* assists *Country 1* by supplying components that provide the *C2* component with extra information. These components consists in *Weather Station*, the *Positioning-B* positioning system, and a *UAV* (Unmanned Aerial Vehicle) drone. However, *C2* cannot use these components directly [23]. Indeed, *Weather Station* that is implemented using CORBA[4], provides specific information such as temperature or humidity whereas *Weather Service*, which is used by *C2*, returns all of this information using a single operation. Further, *Positioning-A* is implemented using SOAP and interacts according to the request/response paradigm whereas *Positioning-B* is implemented using AMQP[5] and hence interacts according to

[3] `http://www.w3.org/TR/soap/`
[4] `http://www.omg.org/technology/documents/corba\_spec\_catalog.htm`
[5] `http://www.amqp.org`

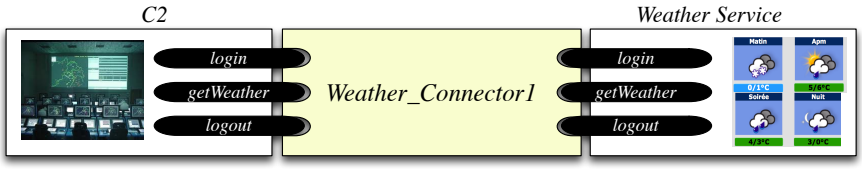

Fig. 2. *C2* and *Weather Service*, and associated connector in *Country 1*

the publish/subscribe paradigm. Also, *UGV* requires the client to login, then it can move in the four cardinal directions while *UAV* is required to takeoff prior to any operation and to land before logging out. To enable *C2* to use the components provided by *Country 2*, with which it is functionally compatible, mediators have to be synthesised and deployed in order to make *C2* interoperate with *Weather Station*, *Positioning-B*, and *UAV*.

3 The Interoperability Problem Space: A Software Architecture Perspective

Software systems may be abstractly described at the architectural level in terms of components and connectors: components are meant to encapsulate computation while connectors are meant to encapsulate interaction. In other words, control originates in components, and connectors are channels for coordinating the control flow (as well as data flow) between components [46].

So as to sustain software composition, software components must not only specify their (provided) interfaces, i.e., the subset of the system's functionality and/or data that is made accessible to the environment, but also the assumptions they make in terms of other components/interfaces that must be present for a component to fulfill its functionality, making all dependencies explicit [49, 24]. A software connector is an architectural element tasked with effecting and regulating interactions among components via their interfaces [49]. As an illustration, Figure 2 depicts the case of the *C2* component interacting with *Weather Service*, where *C2* exhibits an interface in which it requires the operations *login*, *getWeather*, and *logout*, and *Weather Service* defines an interface in which it provides these same operations. The connector *Weather_Connector1* coordinates these operations based on the SOAP middleware technology.

From the standpoint of implementation, middleware provides the adequate basis for implementing connectors. Indeed, middleware greatly eases the composition of components by introducing abstractions that hide the heterogeneity of the underlying infrastructure. However, middleware defines specific data formats and interaction paradigms, making it difficult for components developed using different middleware to communicate [42].

In general, the assembly of components via connectors may conveniently be reasoned about based on the appropriate formalisation of software architecture, as discussed in the following section.

Definitions	
αP	The alphabet of a process P
END	Predefined process. Denotes the state in which a process successfully terminates
set S	Defines a set of action labels
$[i : S]$	Binds the variable i to a value from S
Primitive Processes (P)	
$a \rightarrow P$	Action prefix
$a \rightarrow P \vert b \rightarrow P$	Choice
$P; Q$	Sequential composition
$P(X =' a)$	Parameterised process: P is described using parameter X and modelled for a particular parameter value, $P(a1)$
$P/\{new\_1/old\_1, ..., new\_n/old\_n\}$	Relabelling
$P\backslash\{a_1, a_2, ..., a_n\}$	Hiding
$P + \{a_1, a_2, ..., a_n\}$	Alphabet extension
Composite Processes ($\|P$)	
$P \| Q$	Parallel composition
forall $[i : 1..n]$ $P(i)$	Replicator construct: equivalent to the parallel composition $(P(1) \| ... \| P(n))$.
$a : P$	Process labelling

Table 1. FSP syntax overview

3.1 Formal Foundations for Software Architectures

To enable formal reasoning about software architecture composition, the interaction protocols implemented by components and connectors may be specified using a process algebra, as introduced in the pioneering work of Allen and Garlan [1]. In the context of this chapter, we concentrate more specifically on the use of FSP (Finite State Processes) based on the work of Spitznagel and Garlan, which in particular considers the adaptation of connectors to address dependability as well as interoperability concerns [47].

Finite State Processes. FSP [35] is a process algebra that has proven to be a convenient formalism for specifying concurrent components, analysing, and reasoning about their behaviours. Table 1 provides an overview of the FSP operators, while the interested reader is referred to [35] for further detail. Briefly stated, FSP processes describe actions (events) that occur in sequence, and choices between action sequences. Each process has an alphabet, αP, of the actions that it is aware of (and either engages in or refuses to engage in). There are two types of processes: *primitive processes* and *composite processes*. Primitive processes are constructed through action prefix, choice, and sequential composition. Composite processes are constructed using parallel composition or process relabelling. The replicator forall is a convenient syntactic construct used

to specify parallel composition over a set of processes. Processes can optionally be parameterised and have re-labelling, hiding or extension over their alphabet. A composite process is distinguished from a primitive process by prefixing its definition with ||.

The semantics of FSP is given in terms of Labelled Transition Systems (LTS) [33]. The LTS interpreting an FSP process P can be regarded as a directed graph whose nodes represent the process states and each edge is labelled with an action $a \in \alpha P$ representing the behaviour of P after it engages in a. $P \xrightarrow{a} P'$ then denotes that P transits with action a into P'. Then, $P \overset{s}{\Rightarrow} P'$ is a shorthand for $P \xrightarrow{a_1} P_1 \xrightarrow{a_2} P_2 ... \xrightarrow{a_n} P', s = \langle a_1, a_2, ..., a_n \rangle, a_i \in \alpha P$. There exists a start node from which the process begins its execution. The END state indicates a successful termination. When composed in parallel, processes synchronise on shared actions: if processes P and Q are composed in parallel, actions that are in the alphabet of only one of the two processes can occur independently of the other process, but an action that is in the alphabets of both processes cannot occur until the two of them are willing to engage in it, as described below:

$$\frac{P \xrightarrow{a} P', \nexists a \in \alpha Q}{P \| Q \xrightarrow{a} P' \| Q} \qquad \frac{Q \xrightarrow{a} Q', \nexists a \in \alpha P}{P \| Q \xrightarrow{a} P \| Q'} \qquad \frac{P \xrightarrow{a} P', Q \xrightarrow{a} Q'}{P \| Q \xrightarrow{a} P' \| Q'}$$

Formalising components and connectors using FSP. The interaction protocols run by components are described using a set of FSP processes called *ports*. For example, consider the port of *C2* dedicated to the interaction with *Weather Service* (see Figure 2): *C2* logs in, invokes the operation $getWeather$ several times, and finally logs out. The port of *C2* is specified, using FSP, as follows:

$$\begin{aligned} &C2\_port = (req.login \rightarrow P1), \\ &P1 \quad\quad = (req.getWeather \rightarrow P1 | req.logout \rightarrow C2\_port). \end{aligned}$$

We further use FSP processes to describe a connector as a set of *roles* and a *glue*. Roles are the processes that specify the expected local behaviours of the various interacting parties coordinated by the connector, while the glue process describes the specific coordination protocol that is implemented [1]. Still considering our example of Figure 2, the *Weather_Connector1* connector managing the interactions between *C2* and *Weather Service* defines a role associated with each of them, that is, *C2_role* and *WeatherService_role*, respectively. The connector also defines how these operations are realised using a SOAP request/response paradigm. More specifically, each required operation corresponds to the sending of a SOAP request parameterised with the name of the operation, and the reception of the corresponding SOAP response, which is specified by the process *SOAPClient*. The dual provided operation corresponds to the receiving of a SOAP request parameterised with the name of the operation, and the send of the corresponding SOAP response, which is specified by the process *SOAPServer*. Furthermore, a request sent from one side is received from the other and similarly for a response, which is specified by the process *SOAPGlue*. *Weather_Connector1* is then specified as the parallel composition of all these processes:

$$
\begin{array}{ll}
\textsf{set}\ weather\_actions1 & = \{login, getWeather, logout\} \\
C2\_role & = (req.login \rightarrow P1), \\
P1 & = (req.getWeather \rightarrow P1 | req.logout \rightarrow C2\_Role). \\
 & \\
WeatherService\_role & = (prov.login \rightarrow P2), \\
P2 & = (\ prov.getWeather \rightarrow P2 | prov.logout \rightarrow WeatherService\_role). \\
 & \\
SOAPClient\ (X =' op) & = (req.[X] \rightarrow sendSOAPRequest[X] \rightarrow receiveSOAPResponse[X] \\
 & \quad \rightarrow SOAPClient). \\
SOAPServer\ (X =' op) & = (prov.[X] \rightarrow receiveSOAPRequest[X] \rightarrow sendSOAPResponse[X] \\
 & \quad \rightarrow SOAPServer). \\
SOAPGlue\ (X =' op) & = (sendSOAPRequest[X] \rightarrow receiveSOAPRequest[X] \rightarrow \\
 & \quad sendSOAPResponse[X] \rightarrow receiveSOAPResponse[X] \rightarrow SOAPGlue). \\
 & \\
\| Weather\_Connector1 & = (\ C2\_role \\
 & \quad \| \quad WeatherService\_role \\
 & \quad \| \quad (\textsf{forall}[op : weather\_actions1] SOAPClient(\text{op})) \\
 & \quad \| \quad (\textsf{forall}[op : weather\_actions1] SOAPGlue(\text{op})) \\
 & \quad \| \quad (\textsf{forall}[op : weather\_actions1] SOAPServer(\text{op}))).
\end{array}
$$

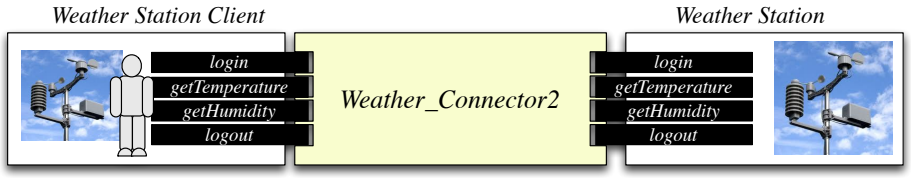

Fig. 3. The *Weather Station* and associated client in *Country 2*

Consider now the case of the *Weather Station* component interacting with its specific client (see Figure 1). As depicted in Figure 3, *Weather Station* exhibits an interface through which it provides the operations *login*, *getTemperature*, *getHumidity*, and *logout*, while the associated port is specified as follows:

$$
\begin{array}{ll}
WeatherStation\_port & = (prov.login \rightarrow P2), \\
P2 & = (\ prov.getTemperature \rightarrow P2 \\
 & \quad |\quad prov.getHumidity \rightarrow P2 \\
 & \quad |\quad prov.logout \rightarrow WeatherStation\_port).
\end{array}
$$

The connector *Weather_Connector2* then coordinates the operations between *Weather Station* and the corresponding client according to the CORBA request/response paradigm and is specified as follows:

$$
\begin{array}{ll}
\textsf{set}\ weather\_actions2 & = \{login, getTemperature, getHumidity, logout\} \\
WeatherStationClient\_role & = (req.login \rightarrow P1), \\
P1 & = (\ req.getTemperature \rightarrow P1 \\
 & \quad |\quad req.getHumidity \rightarrow P1 \\
 & \quad |\quad req.logout \rightarrow WeatherStationClient\_role). \\
WeatherStation\_role & = (prov.login \rightarrow P2), \\
P2 & = (\ prov.getTemperature \rightarrow P2 \\
 & \quad |\quad prov.getHumidity \rightarrow P2 \\
 & \quad |\quad prov.logout \rightarrow WeatherStation\_role).
\end{array}
$$

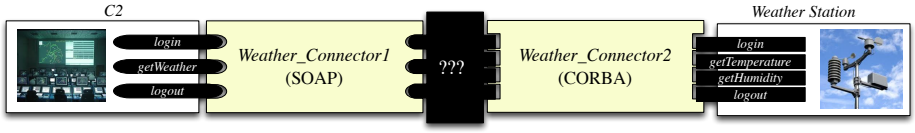

Fig. 4. Architectural mismatch between *C2* and *Weather Station*

$$
\begin{aligned}
&CORBAClient\ (X =' op) &&= (req.[X] \rightarrow sendCORBARequest[X]\\
&&&\quad \rightarrow receiveCORBAResponse[X] \rightarrow CORBAClient).\\
&CORBAServer\ (X =' op) &&= (prov.[X] \rightarrow receiveCORBARequest[X]\\
&&&\quad \rightarrow sendCORBAResponse[X] \rightarrow CORBAServer).\\
&CORBAGlue\ (X =' op) &&= (sendCORBARequest[X] \rightarrow receiveCORBARequest[X]\\
&&&\quad \rightarrow sendCORBAResponse[X] \rightarrow receiveCORBAResponse[X]\\
&&&\quad \rightarrow CORBAGlue).\\
&\|\ Weather\_Connector2 &&= (\ WeatherStationClient\_role\\
&&&\quad \|\ \ WeatherStation\_role\\
&&&\quad \|\ \ (\mathsf{forall}[op : weather\_actions2]\, CORBAClient(\mathrm{op}))\\
&&&\quad \|\ \ (\mathsf{forall}[op : weather\_actions2]\, CORBAGlue(\mathrm{op}))\\
&&&\quad \|\ \ (\mathsf{forall}[op : weather\_actions2]\, CORBAServer(\mathrm{op}))).
\end{aligned}
$$

Thanks to the formal specification of architectural components and connectors, architectural mismatches may be reasoned about. Specifically, architectural mismatches occur when composing two, or more, software components to form a system and those components make different assumptions about their environment [27], thereby preventing interoperability. These assumptions relate to: (i) the data and control models of the involved components, (ii) the protocols and the data model specified by the connector, and (iii) the infrastructure and the development environment on top of which the components are built. Consider for example the composition of *C2* and *Weather Station*. There exists an architectural mismatch between the two, which hampers their interoperation (see Figure 4). Indeed, the components manipulate different data: *C2* deals with weather whereas *Weather Station* manages temperature and humidity. They are also implemented using different middleware technologies: SOAP for *C2* and CORBA for *Weather Station*. In the following section, we show how to reason about the different assumptions that components make about their connection.

3.2 Reasoning about Architectural Mismatches

Architectural mismatches can be reasoned about formally by comparing component port and connector roles [1]. More specifically, a component can be attached to a connector only if its port is *behaviourally compatible* with the connector role it is bound to. Behavioural compatibility between a component port and a connector role is based upon the notion of refinement, i.e., a component port is behaviourally compatible with a connector role if the process specifying the behaviour of the former refines the process characterising the latter [1]. In other words, it should be possible to substitute the role process by the port process.

For example, the *C2* component can be attached to *Weather_Connector1* connector since *C2_port* refines *C2_role* — they are actually the same. Likewise, *WeatherStation_port* refines *WeatherStation_role* defined by

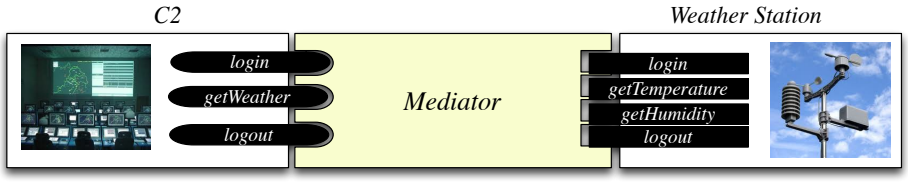

Fig. 5. Mediator to solve architectural mismatch between *C2* and *Weather Station*

Weather_Connector2. However, *WeatherStation_port* cannot be attached to *Weather_Connector1* since it does not refine any of its ports, nor *C2_role* can be attached to *Weather_Connector2*. Hence, in the case of *C2* willing to interact with *Weather Station*, none of the available connectors can readily be used resulting in an architectural mismatch, which needs to be overcome by a mediator, as depicted in Figure 5.

3.3 Mediators Adapting Connectors for Interoperability

In order to solve architectural mismatches without modifying the components themselves, it is necessary to construct a connector that reconciles the assumptions that each of the components makes about its environment. The connector need not necessarily be constructed from scratch. It can also be developed by transforming existing connectors. Hence, a connector with n roles coordinated using a *Glue* process and specified as: $Connector = R_1 \| ... \| R_n \| Glue$ can be adapted into a mediator: $Mediator = f_1(R_1) \| ... \| f_n(R_n) \| R_{n+1} \| ... \| R_{n+k} \| f_G(Glue)$ with which the ports of the components at hand are behaviourally compatible. For example, the mediator between *C2* and *Weather Station* includes roles behaviourally compatible with *C2_port* and *WeatherStation_port* and encompasses the glue that coordinates them.

Spitznagel and Garlan [47] introduce a set of transformation patterns (e.g., data translation and event aggregation), which a developer can specify and compose in order to construct complex connectors based on existing ones. The complex connectors that are specifically considered in [47] enhance the coordination of components with respect to enforcing stronger dependability guarantees. However, complex connectors may as well be built to overcome architectural mismatches. Still, the question that raises itself is which transformations (i.e., composition of the given connector with transformation patterns) are valid and which do not make sense for a specific mismatch at hand. For example, the mediator between *C2* and *Weather Station* has to translate the *getWeather* operation required by *C2* into the *getTemperature* and *getHumidity* operations provided by *Weather Station*. Hence, it needs to compose the connectors *Weather_Connector1* and *Weather_Connector2* , respectively associated with *C2* and the *Weather Station*, with the process:

$$Map = (req.getWeather \rightarrow prov.getTemperature \rightarrow prov.getHumidity).$$

However, this only solves mismatches occurring at the application layer and mediation is also necessary at the middleware layer so as to bridge SOAP and CORBA. Another concern for the composition of connectors is the increasing dynamics of the networking environment, which calls for on-the-fly mediation.

3.4 Dynamic Software Architecture and Mediation

In dynamic environments where components are discovered at runtime and composed dynamically, mediators can no longer be specified or implemented at design time. Instead, they have to be synthesised and deployed on the fly. However, the synthesis of mediators not only requires knowledge of the data and behaviour of the components but also knowledge of the domain, which specifies the relation between the data and operations of the different components. In particular, *ontologies* build upon sound logical theory to provide a machine-interpretable means to reason, automatically, about the semantics of data based on the shared understanding of the domain [4]. As a matter of fact, ontologies prove valuable when dealing with data interoperability. In this context, ontologies are used to specify a shared vocabulary precisely and offer a common basis to reconcile data syntactic differences based on their semantic definitions. They further play a valuable role in software engineering by supporting the automated integration of knowledge among teams and project stakeholders [17]. For example, a weather ontology would allow us to infer the relation between $getWeather$ and $getTemperature$ and $getHumidity$ without a need for human intervention.

Ontologies have also been widely used for the modelling of Semantic Web Services and to achieve efficient service discovery and composition [41, 40]. Semantic Web Services use ontologies as a central point to achieve interoperability between heterogeneous clients and services at runtime. For example, WSMO (Web Service Modelling Ontology) relies on ontologies to support runtime mediation based on pre-defined patterns. However, the proposed approach does not ensure that such mediation does not lead to an erroneous execution (e.g., deadlock) [20]. It further assumes that components are implemented using the same middleware, SOAP.

Overall, the issue of overcoming architectural mismatches to make interoperable components that are functionally compatible, is a cross-cutting concern where protocol mismatches need to be addressed at both application and middleware layers. Interoperability solutions must consider conjointly application and middleware layers: (i) the application layer provides the appropriate level of abstraction to reason about interoperability and automate the generation of mediators; and (ii) the middleware layer offers the necessary services for realising the mediation by selecting and instantiating the specific data structures and protocols. In addition, mediators need to be synthesised on the fly, as the networking environment is now open and dynamic, thereby leading to the assembly of component systems that are known to one another other at runtime, as opposed to design time. As discussed next, supporting such a dynamic cross-layer mediation requires a multifaceted solution.

4 The Interoperability Solution Space: A Multifaceted Review

Sustaining interoperability has received a great deal of attention since the emergence of distributed systems and further promotion of component-based and service-oriented software engineering. We may classify solutions to protocol interoperability according to three broad perspectives: (i) the middleware perspective is specifically concerned with the implementation of middleware-layer mediators, based on the introduction of frameworks that allow bridging components that are implemented on top of heterogeneous infrastructures, (ii) the protocol perspective is focused on the systematic synthesis of mediators based on the specification of the protocols implemented by components to be made interoperable, and (iii) the semantic perspective is oriented toward automated reasoning about the matchmaking of components, both functionally and behaviourally.

4.1 The Middleware Perspective: Implementing Protocol Mediators

By definition, middleware defines an infrastructure mediator that overcomes the heterogeneity occurring in the lower layer. While original middleware solutions primarily targeted data heterogeneity, later middleware solutions had to deal with behavioural heterogeneity due to the composition of component systems relying on heterogeneous middleware protocols, as exemplified by the GMES case study. We then identify two basic approaches for the implementation of protocol mediators: (i) *pairwise mediation* where a specific bridge is implemented for each pair of heterogeneous protocols that need to be composed, and (ii) *mediation through a reference protocol* where protocol interoperability is achieved by bridging any protocol that needs to be composed with a reference protocol. The former leads to highly customised mediators while the latter significantly decreases the development effort associated with mediation. In the following, we describe both approaches in more detail.

Pairwise mediation. Under pairwise mediation, the developer has to define the transformations necessary to reconcile the data and behaviour of the protocols involved and to ensure the correctness of these transformations. Mediation must be addressed at all the layers of protocol heterogeneity. This especially stands for the application and middleware layers, while lower network layer heterogeneity remains largely addressed through IP-based networking. For example, at the middleware layer, OrbixCOMet[6] performs the necessary translation between DCOM and CORBA and SOAP2CORBA[7] ensures interoperability between SOAP and CORBA in both directions.

Figure 6 depicts the example of the composition of *C2* with *Weather Station* using SOAP2CORBA, which allows the SOAP requests issued by *C2* to be

[6] `http://www.iona.com/support/whitepapers/ocomet-wp.pdf`
[7] `http://soap2corba.sourceforge.net/`

Fig. 6. Pairwise mediation between layered protocols

translated into CORBA requests, and the corresponding CORBA responses to be translated into SOAP responses. This translation is relative to a specific operation. Hence, this translation can be specified as follows:

$$SOAP2CORBA(X =' op) = (receiveSOAPRequest[X] \rightarrow sendCORBARequest[X] \rightarrow receiveCORBAResponse[X] \rightarrow sendSOAPResponse[X] \rightarrow SOAP2CORBA).$$

However, the connector *Weather_Connector12_Pairwise* defined as:

$$\begin{aligned} \|Weather\_Connector12\_Pairwise = (\ & C2\_role \\ \| \quad & WeatherStation\_role \\ \| \quad & (\mathsf{forall}[op : weather\_actions1]SOAPClient(\mathrm{op})) \\ \| \quad & (\mathsf{forall}[op : weather\_actions1]SOAPGlue(\mathrm{op})) \\ \| \quad & (\mathsf{forall}[op : weather\_actions1]SOAPServer(\mathrm{op}))). \\ \| \quad & (\mathsf{forall}[op : weather\_actions2]CORBAClient(\mathrm{op})) \\ \| \quad & (\mathsf{forall}[op : weather\_actions2]CORBAGlue(\mathrm{op})) \\ \| \quad & (\mathsf{forall}[op : weather\_actions2]CORBAServer(\mathrm{op}))). \\ \| \quad & SOAP2CORBA \end{aligned}$$

is not a valid connector since the translation is carried out assuming that the components refer to the same application-layer operations to coordinate. For example, when *C2* sends a SOAP request for $getWeather$, it is translated into a CORBA request for $getWeather$, but there is no counter part on the server side since *Weather Station* does not provide this operation. Indeed, higher application-layer mediation also needs to be implemented.

In general, the implementation of pairwise mediators is a complex task: developers have to deal with a lot of details and therefore must have a thorough understanding of the protocols at hand. As a result, solutions that help developers defining middleware-layer mediators have emerged. These solutions consist in a framework whereby the developer provides a declarative specification of the message translation across protocols, based on which the actual transformations are computed. For example, z2z [16] introduces a domain-specific language to describe the protocols to be made interoperable as well as the translation logic to compose them and then generates the corresponding bridge. Starlink [14] uses the same domain-specific models to specify bridges, which it deploys dynamically and interprets at runtime (see Figure 7). However, these solutions still require the developer to specify the translations to be made and hence to know both middleware in advance.

Mediation through a reference protocol. To reduce the development effort induced by pairwise mediation, a reference protocol can be used as an intermediary to translate from one protocol to another. Such mediation is especially

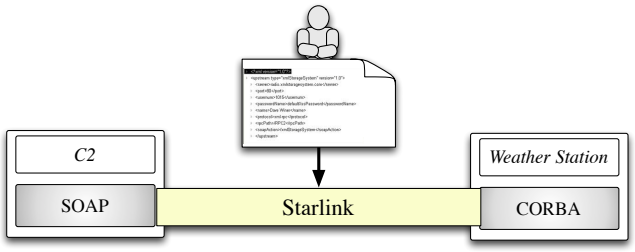

Fig. 7. Pairwise middleware-layer mediation based on high-level specification

appropriate for the middleware layer where heterogeneous middleware protocols may rather easily be mapped onto a common protocol when the middleware implement the same interaction paradigm. Application-layer reference protocols may also be considered for commonly encountered applications like messaging systems [7].

Enterprise Service Buses (ESBs), e.g., Oracle Service Bus[8] and IBM WebSphere Enterprise Service Bus[9], represent the most mature and common use of mediation through a reference protocol. An ESB [39] is an open standard, message-based, distributed integration infrastructure that provides routing, invocation and mediation services to facilitate the interactions of disparate distributed applications and services.

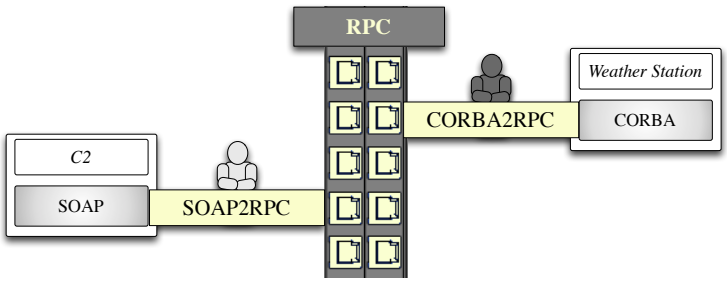

Fig. 8. Mediation through a reference protocol

When the intermediary reference protocol is defined independently of the set of middleware for which it guarantees interoperability, it does not necessarily capture all their details and specificities. Bromberg [12] puts forward the inference of the best-suited intermediary protocol based on the behaviours of the middleware involved. The author applies this approach to ensure interoperability in pervasive systems between different service discovery protocols using

[8] `http://www.oracle.com/technetwork/middleware/service-bus/`
[9] `http://www-01.ibm.com/software/integration/wsesb/`

INDISS [15] and across RPC protocols (assuming the same application atop) using NEMESYS [12].

Going back to our example of the *C2* component willing to interact with *Weather Station*, both of them interact according to the RPC paradigm, which we can use as a reference protocol for ensuring interoperability between SOAP and CORBA (see Figure 8). Hence, we define the RPC (reference) connector as follows:

$$
\begin{array}{ll}
\mathsf{set}\ interface & = \{any\} \\
Client\ (X =' op) & = (sendRequest[X] \rightarrow receiveResponse[X] \rightarrow Client). \\
Server\ (X =' op) & = (receiveRequest[X] \rightarrow sendResponse[X] \rightarrow Server). \\
Glue\ (X =' op) & = (sendRequest[X] \rightarrow receiveRequest[X] \\
 & \quad \rightarrow sendResponse[X] \rightarrow receiveResponse[X] \rightarrow Glue). \\
\|RPCConnector & = (\ (\mathsf{forall}[op : interface]\, Client(\mathrm{op})) \\
 & \quad \|\ \ (\mathsf{forall}[op : interface]\, Glue(\mathrm{op})) \\
 & \quad \|\ \ (\mathsf{forall}[op : interface]\, Server(\mathrm{op}))).
\end{array}
$$

We also need to define the transformations between each protocol and the reference protocol as follows:

$$
\begin{array}{ll}
SOAP2RPC(X =' op) & = (sendSOAPRequest[X] \rightarrow translateSOAP2Request \\
 & \quad \rightarrow sendRequest[X] \rightarrow SOAP2RPC \\
 & \ |\ receiveRequest[X] \rightarrow translate2SOAPRequest \\
 & \quad \rightarrow receiveSOAPRequest[X] \rightarrow SOAP2RPC \\
 & \ |\ sendSOAPResponse[X] \rightarrow translateSOAP2Response \\
 & \quad \rightarrow sendResponse[X] \rightarrow SOAP2RPC \\
 & \ |\ receiveSOAPResponse[X] \rightarrow translate2SOAPResponse \\
 & \quad \rightarrow receiveResponse[X] \rightarrow SOAP2RPC). \\
 & \\
CORBA2RPC(X =' op) & = (sendCORBARequest[X] \rightarrow translateCORBA2Request \\
 & \quad \rightarrow sendRequest[X] \rightarrow CORBA2RPC \\
 & \ |\ receiveRequest[X] \rightarrow translate2CORBARequest \\
 & \quad \rightarrow receiveCORBARequest[X] \rightarrow CORBA2RPC \\
 & \ |\ sendCORBAResponse[X] \rightarrow translateCORBA2Response \\
 & \quad \rightarrow sendResponse[X] \rightarrow CORBA2RPC \\
 & \ |\ receiveCORBAResponse[X] \rightarrow translate2CORBAResponse \\
 & \quad \rightarrow receiveResponse[X] \rightarrow CORBA2RPC).
\end{array}
$$

We obtain the *Weather_Connector12_Reference* connector:

$$
\begin{array}{ll}
\|Weather\_Connector12\_Reference = (\ & C2\_role \\
 & \|\ \ Weather Station\_role \\
 & \|\ \ (\mathsf{forall}[op : weather\_actions1]\, SOAPClient(\mathrm{op})) \\
 & \|\ \ (\mathsf{forall}[op : weather\_actions1]\, SOAP2RPC(\mathrm{op})) \\
 & \|\ \ (\mathsf{forall}[op : weather\_actions2]\, CORBAServer(\mathrm{op})) \\
 & \|\ \ (\mathsf{forall}[op : weather\_actions2]\, CORBA2RPC(\mathrm{op})) \\
 & \|\ \ (\mathsf{forall}[op : weather\_actions1]\, Client(\mathrm{op})) \\
 & \|\ \ (\mathsf{forall}[op : weather\_actions1]\, Glue(\mathrm{op})) \\
 & \|\ \ (\mathsf{forall}[op : weather\_actions1]\, Server(\mathrm{op})) \\
 & \|\ \ (\mathsf{forall}[op : weather\_actions2]\, Client(\mathrm{op})) \\
 & \|\ \ (\mathsf{forall}[op : weather\_actions2]\, Glue(\mathrm{op})) \\
 & \|\ \ (\mathsf{forall}[op : weather\_actions2]\, Server(\mathrm{op}))).
\end{array}
$$

However, as in the case of *Weather_Connector12_Pairwise*, the *Weather_Connector12_Reference* connector only solves interoperability at the middleware layer and must be further enhanced to deal with interoperability at the application layer.

To sum up, a great amount of work exists on the development of concrete interoperability solutions to overcome middleware heterogeneity [9]. All these

approaches tackle middleware interoperability assuming the use of the same application on top, while for components to be able to work together, differences at both application and middleware layers need to be addressed. Similar approaches may be applied for application-layer protocols and actually are, but this is restricted to specific applications that are commonly encountered nowadays, like messaging applications [7]. In general, interoperability solutions based on the implementation of mediators do not scale to the unbounded universe of applications. Another issue is that middleware heterogeneity is often tackled for middleware defining the same interaction paradigm, while systems envisioned for the Future Internet are increasingly heterogeneous and require to compose systems based on distinct paradigms. The notion of extensible service buses enabling highly heterogeneous systems to interoperate across interaction paradigms has recently emerged but it is in its infancy [28]. The provision of interoperability solutions remains, however, a complex task for which automated support is of a great help.

4.2 The Protocol Perspective: Synthesising Protocol Mediators

In order to ease the task of the developers in achieving interoperability between functionally-compatible components, one approach is to provide methods and tools for the automated synthesis of mediators based on the specification of the protocols involved. The approaches can be applied at the application and middleware layers as long as they are isolated.

Lam [34] defines an approach for the synthesis of mediators using a reference protocol, which represents the glue of the mediator. Developers define the reference protocol based on an intuitive understanding of the features common to the protocols at hand. The author defines an approach for computing the relabelling function that maps the individual protocol onto the reference protocol. The mediator is then composed of the relabelling functions together with the reference protocol. As discussed in Section 4.1, at the middleware layer, we can specify the following reference protocol, which represents the glue of an RPC protocol:

$$\begin{aligned} \|\mathit{Reference\_Middleware\_protocol}\ (X =' op) = (&\mathit{sendRequest}[X] \rightarrow \mathit{receiveRequest}[X] \\ &\rightarrow \mathit{sendResponse}[X] \rightarrow \mathit{receiveResponse}[X] \\ &\rightarrow \mathit{Reference\_Mdw\_protocol}). \end{aligned}$$

However at the application layer, the synthesis cannot be applied as relabelling involves the translation of one operation only and not a sequence of operations. That is *getWeather*, *getTemperature*, and *getHumidity* cannot be mapped to the same operation. Hence, the *Weather_Connector12_Synthesised* connector does not allow *C2* and *WeatherStation* to interoperate:

$$\begin{aligned} \|\mathit{Weather\_Connector12\_Synthesised} = (\ & \mathit{C2\_role} \\ \|\ & \mathit{WeatherStation\_role} \\ \|\ & (\mathsf{forall}[op : \mathit{weather\_actions1}]\mathit{SOAPClient}(\mathrm{op})) \\ & /\{\mathit{sendSOAPRequest}/\mathit{sendRequest}, \\ \|\ & \ \ \mathit{receiveSOAPResponse}/\mathit{receiveResponse}\} \\ \|\ & (\mathsf{forall}[op : \mathit{weather\_actions2}]\mathit{CORBAServer}(\mathrm{op})) \\ & /\{\mathit{receiveCORBARequest}/\mathit{receiveRequest}, \\ & \ \ \mathit{sendCORBAResponse}/\mathit{sendResponse}\} \\ \|\ & (\mathsf{forall}[op : \mathit{weather\_actions1}]\mathit{Glue}(\mathrm{op})) \end{aligned}$$

The aforementioned solution consists in defining an abstraction common to the protocols to be mediated. An alternative approach is to synthesise a pairwise mediator based on a partial specification of the translations to be made, either as a goal, which represents a global specification of the composed system, or as an interface mapping, which represents the correspondence between the operations of the components. In the case of interoperability between *C2* and *Weather Station*, consider for example the following goal:

$$
\begin{array}{l}
Req1 = (Country1.req.login \rightarrow Country2.prov.login \rightarrow Req1).\\
Req2 = (Country1.req.getWeather \rightarrow P1),\\
P1 \quad = (Country2.prov.getTemperature \rightarrow Country2.prov.getHumidity \rightarrow Req2\\
\qquad \mid\ Country2.prov.getHumidity \rightarrow Country2.prov.getTemperature \rightarrow Req2).\\
Req3 = Country1.req.logout \rightarrow Country2.prov.logout \rightarrow Req3).\\
\textsf{property}\ \|Goal = (Req1 \parallel Req2 \parallel Req3).
\end{array}
$$

Note that the actions are prefixed with either *Country1* or *Country2* so as to prevent synchronisations outside the mapping processes. The goal ensures that each time *C2* performs a *login* or a *logout*, then *Weather Station* eventually performs it as well. When *C2* issues a request for $getWeather$, then *Weather Station* eventually provides $getTemperature$ and $getHumidity$ in any order. Calculating the mediator amounts to computing the process M, which refines the composition ($C2 \parallel WeatherStation$) so as to satisfy the *Goal* property.

Calvert and Lam [18] propose to calculate the composition first, then to eliminate the traces that violate the goal. Applied to our running example, first the composition ($C2 \parallel WeatherStation \parallel Goal$) is calculated, then all the traces where the goal cannot be satisfied are removed, which results in the most general mediator called *quotient*. However, this calculation is computationally expensive as it requires covering all the trace set. In order to eliminate execution errors (e.g., deadlocks) efficiently, model checking can be used in the generation of mediators [8, 19, 37].

To avoid the reliance on model checking techniques, Yellin and Strom [53] propose an algorithm for the automated synthesis of mediators based on a declarative interface mapping. The authors assume a non-ambiguous one-to-one interface mapping, i.e., an operation corresponds to one operation only. They construct the mediator by exploring the protocols of the mediator and performing the necessary translations so as to guarantee that no deadlock can happen.

All the aforementioned approaches expect the transformations to be partially specified. In other words, the mediation problem has been shifted to the goal or interface mapping definition. Most of the difficulty remains on the definition of the partial specification, which require developers to know the protocols of both components and to have an intuitive understanding of the translations that need to be performed to enable them to interoperate. Given the size and the number of parameters of the interface of each component, this task may be error-prone and perhaps as difficult as providing the mediator itself. For example, the Amazon Web Service[10] includes 23 operations and no less than 72 data type definitions and eBay[11] contains more than 156 operations. Given all possible

[10] `http://soap.amazon.com/schemas2/AmazonWebServices.wsdl`

[11] `http://developer.ebay.com/webservices/latest/ebaysvc.wsdl`

combinations, methods that automatically compute this partial specification are necessary, which we survey next.

4.3 The Semantic Perspective: Emergent Protocol Mediators

Ontologies provide experts with a means to formalise the knowledge about domains as a set of axioms that make explicit the intended meaning of a vocabulary [30]. Hence, besides general purpose ontologies, such as dictionaries (e.g., WordNet[12]) and translators (e.g., BOW[13]), there is an increasing number of ontologies available for various domains such as biology [3], geoscience [44], and social networks [29], which in turn foster the development of a multitude of search engines for finding ontologies on the Web [25].

Ontologies are supported by a logic theory to reason about the properties and relations holding between the various domain entities. In particular, OWL[14] (Web Ontology Language), which is the W3C standard language to model ontologies, is based on Description Logics (DL). While traditional formal specification techniques (e.g., first-order logic) might be more powerful, DL offers crucial advantages: it excels at modelling domain-specific knowledge while providing decidable and efficient reasoning algorithms. DL is used to formally specify the vocabulary of a domain in terms of concepts, features of each concept, and relationships between these concepts [26]. DL also allows the definition of complex types out of primitive ones, is able to detect specialisation relations between complex types, and to test the consistency of types. Traditionally, the basic reasoning mechanism in DL is *subsumption*, which can be used to implement other inferences (e.g., satisfiability and equivalence) using pre-defined reductions [4]. In this sense, DL in many ways resembles type systems with some inference mechanisms such as subsumption between concepts and classification of instances within the appropriate concept, corresponding to type subsumption and type inference respectively. Nevertheless, DL is by design and tradition well-suited for application- and domain-specific services [11].

Besides defining the semantics of data, OWL-S [36] adds the definition of the *capability* of a service, which defines the service's functionality, and the service's *process model*, which defines how this functionality is performed. Services can then be matched based on their capabilities [43] or based on their process model. In this latter case, Vaculín *et al.* [51] devise a mediation approach for OWL-S processes. They first generate all requesters' paths, then find the appropriate mapping for each path by simulating the provider process. This approach deals only with client/server interactions and is not able to generate a mediator if many mappings exist for the same operation. However, OWL-S only has a qualified consent because it specifies yet another model to define services. In addition, solutions based on process algebra and automata have proven to be more suitable for reasoning about protocol interoperability.

[12] `http://www.w3.org/TR/wordnet-rdf/`
[13] `http://BOW.sinica.edu.tw/`
[14] `http://www.w3.org/TR/owl2-overview/`

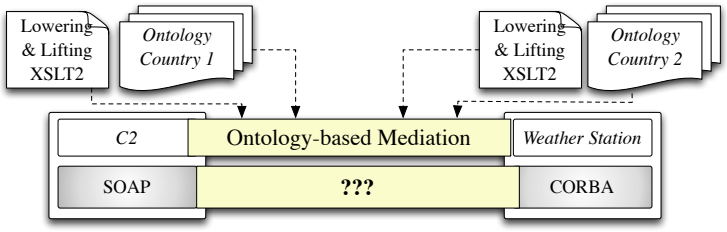

Fig. 9. Ontology-based mediation

In this direction, WSMO [20] defines a description language that integrates ontologies with state machines for representing Semantic Web Services. However, these states machines are not used to synthesise mediators. Instead, a runtime mediation framework, the Web Service Execution Environment (WSMX) mediates interaction between heterogeneous services by inspecting their individual protocols and perform the necessary translation on the basis of pre-defined mediation patterns while the composition of these patterns is not considered, and there is no guarantee that it will not lead to a deadlock.

Considering again our GMES example, with the knowledge of the weather domain encoded within a weather ontology, it can be inferred that the *getWeather* operation required by *C2* corresponds to the *getTemperature* and *getHumidity* operations provided by *Weather Station*. As a result, the following mapping process can be generated:

$$Map = (req.getWeather \rightarrow prov.getTemperature \rightarrow prov.getHumidity).$$

We obtain the following connector:

$$\begin{aligned} \| \, Weather\_Connector12\_Semantic = (\ & C2\_role \\ \| \quad & WeatherStation\_role \\ \| \quad & (\mathsf{forall}[op : weather\_actions1]SOAPClient(\mathrm{op})) \\ \| \quad & (\mathsf{forall}[op : weather\_actions1]SOAPGlue(\mathrm{op})) \\ \| \quad & (\mathsf{forall}[op : weather\_actions1]SOAPServer(\mathrm{op}))) \\ \| \quad & (\mathsf{forall}[op : weather\_actions2]CORBAClient(\mathrm{op})) \\ \| \quad & (\mathsf{forall}[op : weather\_actions2]CORBAGlue(\mathrm{op})) \\ \| \quad & (\mathsf{forall}[op : weather\_actions2]CORBAServer(\mathrm{op}))). \\ \| \quad & Map \end{aligned}$$

However, *C2* and *Weather Station* cannot interact successfully through *Weather_Connector12_Semantic* since the coordination at the middleware layer is not performed. For example, it is not specified how to coordinate the *getWeather* SOAP request with the *getTemperature* and *getHumidity* CORBA requests (see Figure 9).

The need for ontologies to achieve interoperability is not specific to the Web Service domain but should be considered for highly heterogeneous environments where components may be built using diverse middleware technologies. It is in particular worth highlighting the consensus that ontologies are key to the IoT vision [50]. As a result, it is indispensable to combine appropriate techniques to handle the multifaceted nature of interoperability. These techniques include formal approaches for the synthesis of mediators with support of ontology-based

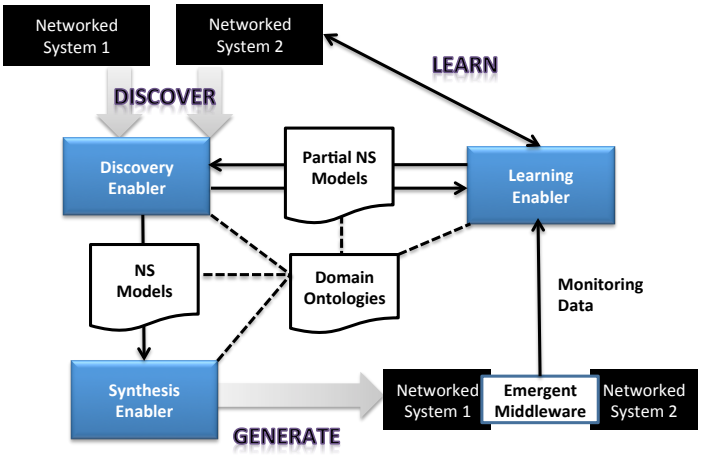

Fig. 10. The CONNECT architecture for the realisation of emergent middleware [10]

reasoning so as to automate the synthesis, together with middleware solutions to realise and execute these mediators and enable components to interoperate effectively. We have investigated such a multifaceted solution to interoperability within the CONNECT project [10].

5 Emergent Middleware: A Multifaceted Approach to Interoperability

In this section, we present the solution elaborated in the context of the European CONNECT project that revolves around the notion of emergent middleware and related enablers so as to sustain interoperability in the increasingly connected digital world. An emergent middleware is a dynamically generated distributed system infrastructure for the current operating environment and context, which allows functionally-compatible systems to interoperate seamlessly.

5.1 Emergent Middleware Enablers

In order to produce an emergent middleware, an architecture of *Enablers* is required that support the realisation of mediators into emergent middleware. An Enabler is a software component responsible for a specific step in the realisation of emergent middleware and which coordinates with other Enablers during this process.

As depicted in Figure 10, the emergent middleware Enablers are informed by *domain ontologies* that formalise the concepts associated with the application domains (i.e., the vocabulary of the application domains and their relationships) of interest as well as with middleware solutions (i.e., the vocabulary defining

middleware peculiarities, from interaction paradigms to related messages). Three *Enablers*, which are presented below, must then be comprehensively elaborated to fully realise emergent middleware.

Discovery Enabler: The *Discovery Enabler* is in charge of finding the components operating in a given environment. The Discovery Enabler receives both the advertisement messages and lookup request messages that are sent within the network environment by the components using legacy discovery protocols (e.g., SLP[15], WS-Discovery[16], UPnP-SSDP[17], Jini[18]). The Enabler obtains this input by listening on known multicast addresses (used by legacy discovery protocols), as common in interoperable service discovery [15]. These messages are then processed, using plug-ins associated with legacy discovery protocols, thereby allowing to extract basic component models from the information exposed by the components, i.e., identification of the components' interfaces together with middleware used for interactions. We build upon the ontology-based modelling as defined by Semantic Web Services (presented in Section 4.3) to model components. The model of a component includes: (i) a semantic description of the functionality it requires or provides, that is, its capability, (ii) a description of the interface of the component, which is augmented with ontology-based annotations attached to the operations required or provided by the component, (iii) a description of the interaction protocol run by the component, that is behaviour, and (iv) a specific middleware used to implement this behaviour and further refine the execution of operations. For example, as illustrated in Section 3.1, in the case of a SOAP middleware, a required operation corresponds to the sending of a SOAP request parameterised with the name of the operation, and the reception of the corresponding SOAP response. In a dual manner, a provided operation corresponds to the reception of a SOAP request parameterised with the name of the operation, and the sending of the corresponding SOAP response. However, using existing discovery protocols, components only expose their syntactic interfaces. Hence, the Discovery Enabler relies on the Learning Enabler to complete the model of a component.

Learning Enabler: The *Learning Enabler* uses advanced learning algorithms to dynamically infer the ontology-based semantics of the component's capability and operations, as well as determine the behaviour of a component, given the interface description it exposes through some legacy discovery protocol. The Learning Enabler implements both statistical and automata learning to feed component models with adequate semantic knowledge, i.e., functional and behavioural semantics, respectively [6]. The Learning Enabler must interact directly with a component in order to learn its behaviour.

[15] `http://www.openslp.org/`
[16] `http://docs.oasis-open.org/ws-dd/discovery/1.1/os/wsdd-discovery-1.1-spec-os.pdf`
[17] `http://www.upnp.org/`
[18] `http://www.jini.org/`

Synthesis Enabler: Once component models are complete, initial semantic matching of their capabilities is performed to determine whether or not the components are functionally compatible [43]. The automated synthesis of mediators between functionally compatible components, which is the main role of the *Synthesis Enabler*, lies at the heart of the realisation of the emergent middleware. It puts together the various perspectives on interoperability presented in Section 4 so as to provide a unified solution to the automated generation of mediators and their implementation as emergent middleware.

The semantic perspective provides us with tools to compute the interface mapping automatically by using domain ontologies in order to reason about the semantics of the required and provided operations of the components and to infer the semantic correspondence between them. More specifically, we first define the conditions under which a sequence of required operations can be mapped to a sequence of provided operations. These conditions state that (i) the functionality offered by the provided operations covers that of the required ones, (ii) each provided operation has its input data available (in the right format) at the time of execution, and (iii) each required operation has its output data available (also in the appropriate format) at the time of execution. Then, we use constraint programming, which we leverage to support ontology reasoning, in order to compute the interface mapping efficiently [21].

The protocol perspective provides us with the foundations for synthesising mediators based on the generated interface mapping. More specifically, we define an approach that uses interface mapping to build the mediator incrementally by forcing the protocols at hand to progress consistently so that if one requires a sequence of operations, the interacting process is ready to engage in a sequence of provided operations to which it maps according to the interface mapping. Given that an interface mapping guarantees the semantic compatibility between the operations of the components, then the mediator synchronises with both protocols and compensates for the differences between their actions by performing the necessary transformations. The mediator further consumes the extra output actions so as to allow protocols to progress. The synthesis of mediators deals only with required and provided operations, while their actual implementation is managed by specific middleware [31].

Finally, the middleware perspective provides the background necessary to implement mediators and turn them into emergent middleware. In particular, we build upon the approach of a pairwise-mediation framework which, given a specification of the translations that need to be made, deploys the mediator and executes the necessary translations to make functionally compatible components interoperate. Hence, it suffices to provide the mediator previously synthesised as input to the mediation framework.

Adaptive Emergent Middleware: The Learning phase is a continuous process where the knowledge about components is enriched over time, thereby implying that emergent middleware possibly needs to adapt as the knowledge evolves. The synthesised emergent middleware is equipped with monitoring probes that gather information on actual interaction between connected systems. This ob-

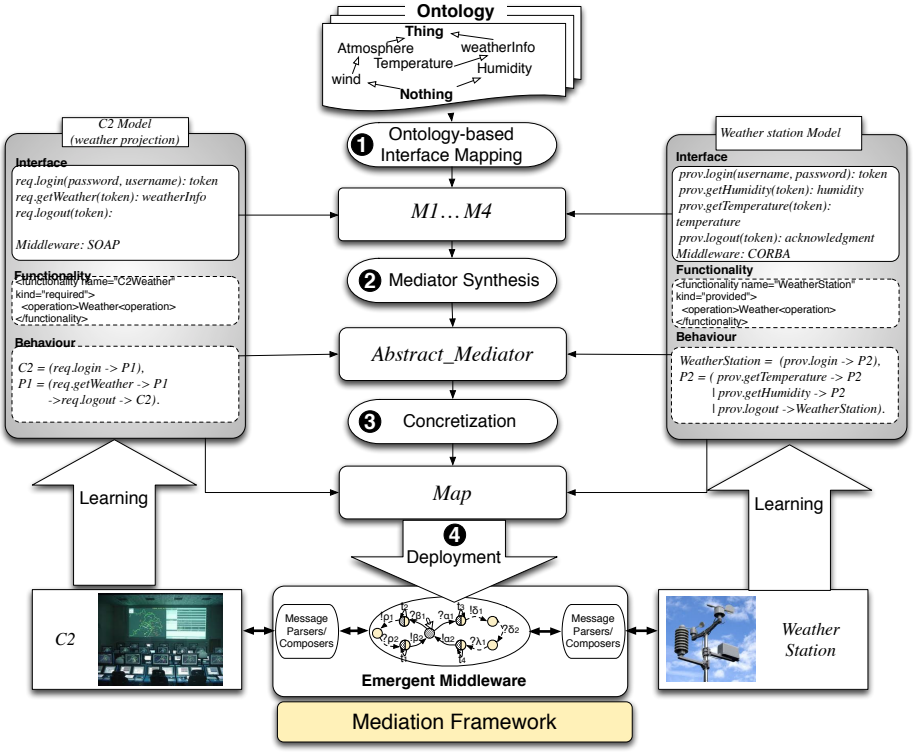

Fig. 11. Emergent middleware between *C2* and *Weather Station*

served *Monitoring Data* (see Figure 10) is delivered to the Learning Enabler, where the learned hypotheses about the components' behaviour are compared to the observed interactions. Whenever an observation is made by the monitoring probes that is not contained in the learned behavioural models, another iteration of learning is triggered, yielding refined behavioural models. These models are then used to synthesise and deploy an updated emergent middleware.

5.2 Emergent Middleware in GMES

Figure 11 depicts the steps to produce the emergent middleware that makes *C2* and *Weather Station* interoperate. The models of *C2* and *Weather Station* can be automatically inferred from their discovered interface as detailed in [6]. In this section, we focus on the steps for synthesising the mediator that ensures interoperability between *C2* and *Weather Station*.

Using a weather ontology, we calculate the interface mapping (see Figure 11-❶), which results in the definition of the following processes:

$$
\begin{aligned}
M1 &= (Country1.req.login \rightarrow Country2.prov.login \rightarrow \mathsf{END}).\\
M2 &= (Country1.req.getWeather \rightarrow Country2.prov.getTemperature\\
&\quad \rightarrow Country2.prov.getHumidity \rightarrow \mathsf{END}).\\
M3 &= (Country1.req.getWeather \rightarrow Country2.prov.getHumidity\\
&\quad \rightarrow Country2.prov.getTemperature \rightarrow \mathsf{END}).\\
M4 &= (Country1.req.logout \rightarrow Country2.prov.logout \rightarrow \mathsf{END}).
\end{aligned}
$$

The abstract mediator *Abstract_Map* coordinates these processes in order for the composition (*C2* $\parallel$ *Abstract_Map* $\parallel$ *WeatherStation*) to be free from deadlocks. When translating the *getWeather* operation required by *C2*, both $M2$ and $M3$ are applicable but we have to choose only one of them as the mediator cannot perform internal choice (see Figure 11-❷). The abstract mediator is as follows:

$$
\begin{aligned}
Abstract\_Mediator &= (Country1.req.login \rightarrow Country2.prov.login \rightarrow AMap),\\
AMap &= (Country1.req.getWeather \rightarrow Country2.prov.getTemperature\\
&\quad \rightarrow Country2.sendCORBARequest.getHumidity \rightarrow AMap\\
&\quad \mid Country1.req.logout \rightarrow Country2.prov.logout \rightarrow Abstract\_Mediator).
\end{aligned}
$$

We concretise the abstract mediator by taking into account the middleware used by each component. For example, the SOAP request for the login operation received from *C2* is translated to a CORBA request for the login operation and forwarded to *Weather Station*. Then, the associated CORBA response received from *Weather Station* is transformed to a SOAP response and sent to *C2*. A SOAP request for a *getWeather* is translated to a CORBA request for *getTemperature* together with another CORBA request for *getHumidity*. Then the CORBA responses for *getTemperature* and *getHumidity* are translated into a SOAP response for *getWeather* [5]. The resulting Map process is as follows (see Figure 11-❸):

$$
\begin{aligned}
Map &= (Country1.receiveSOAPRequest.login \rightarrow Country2.sendCORBARequest.login\\
&\quad \rightarrow Country2.receiveCORBAResponse.login \rightarrow Country1.sendSOAPResponse.login\\
&\quad \rightarrow Map1),\\
Map1 &= (Country1.receiveSOAPRequest.getWeather\\
&\quad \rightarrow Country2.sendCORBARequest.getTemperature\\
&\quad \rightarrow Country2.receiveCORBAResponse.getTemperature\\
&\quad \rightarrow Country2.sendCORBARequest.getHumidity\\
&\quad \rightarrow Country2.receiveCORBAResponse.getHumidity\\
&\quad \rightarrow Country1.sendSOAPResponse.getWeather\\
&\quad \rightarrow Map1\\
&\quad \mid Country1.receiveSOAPRequest.logout \rightarrow Country2.sendCORBARequest.logout\\
&\quad \rightarrow Country2.receiveCORBAResponse.logout \rightarrow Country1.sendSOAPResponse.logout\\
&\quad \rightarrow Map).
\end{aligned}
$$

Finally, the mediator is deployed on top of a mediation framework, Starlink [13], which executes the Map process and generates parsers and composers to deal with middleware-specific messages (see Figure 11-❹). The resulting connector *Weather_Mediator* make *C2* and *Weather Station* interoperate:

$$
\begin{aligned}
\parallel Weather\_Mediator = (\ & Country1 : C2\_role\\
\parallel\ & Country2 : WeatherStation\_role\\
\parallel\ & (\mathsf{forall}[op : weather\_actions1]Country1 : SOAPClient(\mathrm{op}))\\
\parallel\ & (\mathsf{forall}[op : weather\_actions1]Country1 : SOAPGlue(\mathrm{op}))\\
\parallel\ & (\mathsf{forall}[op : weather\_actions1]Country1 : SOAPServer(\mathrm{op}))\\
\parallel\ & (\mathsf{forall}[op : weather\_actions2]Country2 : CORBAClient(\mathrm{op}))\\
\parallel\ & (\mathsf{forall}[op : weather\_actions2]Country2 : CORBAGlue(\mathrm{op}))\\
\parallel\ & (\mathsf{forall}[op : weather\_actions2]Country2 : CORBAServer(\mathrm{op}))).\\
\parallel\ & Map).
\end{aligned}
$$

To sum up, this simple example allows us to illustrates the CONNECT approach to the synthesis of emergent middleware in order to achieve interoperability between components that feature differences at both the application and middleware layers. Ontologies play a crucial role in this process by allowing us to reason about the meaning of information exchanged between components and infer the mappings necessary to make them operate together. Likewise, behavioural analysis enables us to synthesise the mediator that coordinates the components' behaviours and guarantees their successful interaction. Finally, middleware technologies allow us to enact the mediator through the concept of emergent middleware. Nevertheless, to enable automated reasoning about interoperability and the generation of appropriate mediators, we focus on the ontological concepts, which represent the types of the input/output data exchanged between components. However, there are situations where reasoning about the value of the data is also necessary. For example, to be able to ensure interoperability between components using different streaming protocols, the mediator is required to deal with lower-level details such as the appropriate encoding and the segmentation of data. In this particular case, the mediator can be partially specified at design-time and deployed at runtime, as is the case for AmbiStream [2].

The accuracy of the components' models may also impact the emergent middleware. While machine learning significantly improves automation by inferring the model of the component from its implementation, it also induces some inaccuracy that may lead the emergent middleware to reach an erroneous state. Hence, the system needs to be continuously monitored so as to evaluate the correspondence between the actual system and its model. In the case where the model of one of the components changes, then the mediator should be updated accordingly in order to reflect this change. Another imprecision might also be due to ontology alignment. Hence, incremental re-synthesis would be very important to cope with both the dynamic aspect and partial knowledge about the environment.

6 Conclusion

In spite of the extensive research effort, interoperability remains an open and critical challenge for today's and even more tomorrow's highly heterogeneous and dynamic networking environments. This chapter has surveyed state-of-the-art approaches to interoperability, highlighting the multiple perspectives that need to be considered and which span: (i) middleware-layer implementation so as to provide abstractions hiding the heterogeneity of the environment, (ii) protocol synthesis so as to relieve as much as possible the developers in dealing with the implementation of custom mediators that must overcome heterogeneity from the application down to the middleware layers, and (iii) ontology-based specification of system models so as to allow fully automated mediator synthesis that is a key requirement of the dynamic networking environment. The chapter has then outlined the CONNECT approach to interoperability, which unifies these different perspectives so as to enable interoperability in a future-proof manner. CONNECT

specifically advocates a solution based on maintaining a sophisticated model of the system at runtime [5]. This includes capturing aspects related to the components' capabilities, interfaces, associated data and behaviour. The solution is then supported by a range of enablers that capture or act on this information to enable the runtime realisation of emergent middleware between given components, i.e., protocol mediators that reconcile the discrepancies occurring in both application- and middleware-layer protocols. The end result can be seen as a two dimensional space related to (i) the meta-information that is captured about the system, and (ii) the associated middleware functions that operate on this meta-information space. This is then supported by ontologies that provide meaning and reasoning capabilities across all enablers and aspects of meta-information known about the system.

It is important to stress that interoperability is, as with many features of distributed systems, an end-to-end problem. For example, it is not sufficient to achieve interoperability between application-level interfaces. Rather, interoperability can only be achieved through a coordinated approach involving application, middleware and underlying network levels so that components can interoperate in spite of potential heterogeneity in descriptions, in middleware deployments and network environments they operate in. And, this needs to be achieved dynamically according to the current context, assuming contexts may change, thereby requiring self-adaptive emergent middleware. More generally, future work includes examining the application of the CONNECT approach to deal with uncontrolled changes in the environment, and expanding the scope of the work to include non-functional concerns associated with communication instances (including performance, dependability and security properties). There is also considerable potential for core research on emergent middleware in areas such as the role of probabilistic reasoning in order to support uncertainties in the ontology, the possibility of learning new ontological information as it becomes available, and also dealing with heterogeneity in the ontologies.

Acknowledgments.

This work is carried out as part of the European FP7 ICT FET CONNECT (`http://connect-forever.eu/`) project.

References

1. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Trans. Softw. Eng. Methodol. (1997)
2. Andriescu, E., Cardoso, R.S., Issarny, V.: Ambistream: A middleware for multimedia streaming on heterogeneous mobile devices. In: Proc. of Middleware. pp. 249–268 (2011)
3. Aranguren, M., Bechhofer, S., Lord, P., Sattler, U., Stevens, R.: Understanding and using the meaning of statements in a bio-ontology: recasting the gene ontology in OWL. BMC bioinformatics 8(1), 57 (2007)

4. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F.: The Description Logic Handbook. Cambridge University Press (2003)
5. Bencomo, N., Bennaceur, A., Grace, P., Blair, G., Issarny, V.: The role of models@run.time in supporting on-the-fly interoperability. Springer Journal on Computing (2012)
6. Bennaceur, A., Issarny, V., Sykes, D., Howar, F., Isberner, M., Steffen, B., Johansson, R., Moschitti, A.: Machine learning for emergent middleware. In: Proc. of the Joint workshop on Intelligent Methods for Soft. System Eng., JIMSE (2012)
7. Bennaceur, A., Issarny, V., Spalazzese, R., Tyagi, S.: Achieving interoperability through semantics-based technologies: The instant messaging case. In: 11th International Semantic Web Conference, ISWC (2012)
8. Bersani, M., Cavallaro, L., Frigeri, A., Pradella, M., Rossi, M.: SMT-based verification of ltl specification with integer constraints and its application to runtime checking of service substitutability. In: Software Engineering and Formal Methods (SEFM), 2010 8th IEEE International Conference on. pp. 244–254. IEEE (2010)
9. Blair, G., Paolucci, M., Grace, P., Georgantas, N.: Interoperability in complex distributed systems. In: Bernardo, M., Issarny, V. (eds.) SFM-11: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems – Connectors for Eternal Networked Software Systems. Springer Verlag (2011)
10. Blair, G.S., Bennaceur, A., Georgantas, N., Grace, P., Issarny, V., Nundloll, V., Paolucci, M.: The role of ontologies in emergent middleware: Supporting interoperability in complex distributed systems. In: Proc. of Middleware (2011)
11. Borgida, A.: From type systems to knowledge representation: Natural semantics specifications for description logics. Int. J. Cooperative Inf. Syst. 1(1), 93–126 (1992)
12. Bromberg, Y.D.: Solutions to middleware heterogeneity in open networked environment. Ph.D. thesis, Université de Versailles Saint-Quentin-en-Yvelynes (2006)
13. Bromberg, Y.D., Grace, P., Réveillère, L.: Starlink: runtime interoperability between heterogeneous middleware protocols. In: International Conference on Distributed Computing Systems, ICDCS (2011)
14. Bromberg, Y.D., Grace, P., Réveillère, L., Blair, G.S.: Bridging the interoperability gap: Overcoming combined application and middleware heterogeneity. In: Proc. of Middleware. pp. 390–409 (2011)
15. Bromberg, Y.D., Issarny, V.: INDISS: Interoperable discovery system for networked services. In: Proc. of Middleware (2005)
16. Bromberg, Y.D., Réveillère, L., Lawall, J.L., Muller, G.: Automatic generation of network protocol gateways. In: Proc. of Middleware (2009)
17. Calero, C., Ruiz, F., Piattini, M.: Ontologies for Software Engineering and Software Technology. Springer-Verlag (2006)
18. Calvert, K.L., Lam, S.S.: Deriving a protocol converter: A top-down method. In: Proc. of the Symposium on Communications Architectures & Protocols, SIGCOMM. pp. 247–258 (1989)
19. Cavallaro, L., Nitto, E.D., Pradella, M.: An automatic approach to enable replacement of conversational services. In: Proc. of the 7th International Joint Conference on Service-Oriented Computing, ICSOC/ServiceWave (2009)
20. Cimpian, E., Mocan, A.: WSMX process mediation based on choreographies. In: Proc. of Business Process Management Workshop (2005)
21. Connect Consortium: Connect Deliverable D3.3: Dynamic connector synthesis: revised prototype implementation. FET IP Connect EU project., `http://hal.inria.fr/hal-00695592/`

22. CONNECT Consortium: CONNECT Deliverable D6.3: Experiment scenarios, prototypes and report - Iteration 2. FET IP CONNECT EU project., http://hal.inria.fr/hal-00695639
23. CONNECT Consortium: CONNECT Deliverable D6.4: Assessment report: Experimenting with CONNECT in Systems of Systems, and Mobile Environments. FET IP CONNECT EU project., http://hal.inria.fr/hal-00793920
24. Coulouris, G.F., Dollimore, J., Kindberg, T., Blair, G.: Distributed systems: concepts and design, Fifth Edition. Addison-Wesley Longman (2012)
25. d'Aquin, M., Noy, N.F.: Where to publish and find ontologies? a survey of ontology libraries. J. Web Sem. 11, 96–111 (2012)
26. Dong, J.S.: From semantic web to expressive software specifications: a modeling languages spectrum. In: Proc. of the International Conference on Software Engineering, ICSE (2006)
27. Garlan, D., Allen, R., Ockerbloom, J.: Architectural mismatch or why it's hard to build systems out of existing parts. In: International Conference on Software Engineering, ICSE (1995)
28. Georgantas, N., Rahaman, M.A., Ameziani, H., Pathak, A., Issarny, V.: A coordination middleware for orchestrating heterogeneous distributed systems. In: 6th International Conference on Advances in Grid and Pervasive Computing, GPC. pp. 221–232 (2011)
29. Golbeck, J., Rothstein, M.: Linking social networks on the web with foaf: A semantic web case study. In: AAAI. pp. 1138–1143 (2008)
30. Guarino, N.: Helping people (and machines) understanding each other: The role of formal ontology. In: CoopIS/DOA/ODBASE (1). p. 599 (2004)
31. Issarny, V., Bennaceur, A., Bromberg, Y.D.: Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability. In: Bernardo, M., Issarny, V. (eds.) SFM-11: 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems – Connectors for Eternal Networked Software Systems. Springer Verlag (2011)
32. Jamshidi, M.: Systems of systems engineering: principles and applications. CRC Press (2008)
33. Keller, R.M.: Formal verification of parallel programs. Communications of the ACM 19(7), 371–384 (1976)
34. Lam, S.S.: Protocol conversion. IEEE Transaction Software Engineering (1988)
35. Magee, J., Kramer, J.: Concurrency : State models and Java programs. Hoboken (N.J.) : Wiley (2006)
36. Martin, D.L., Burstein, M.H., McDermott, D.V., McIlraith, S.A., Paolucci, M., Sycara, K.P., McGuinness, D.L., Sirin, E., Srinivasan, N.: Bringing semantics to web services with owl-s. In: Proc. if the World Wide Web conference, WWW'07. pp. 243–277 (2007)
37. Mateescu, R., Poizat, P., Salaün, G.: Adaptation of service protocols using process algebra and on-the-fly reduction techniques. IEEE Trans. Software Eng. 38(4), 755–777 (2012)
38. McIlraith, S.A., Son, T.C., Zeng, H.: Semantic web services. IEEE Intelligent Systems 16(2), 46–53 (2001)
39. Menge, F.: Enterprise Service Bus. In: Proc. of the Free and open source soft. conf. (2007)
40. Mokhtar, S.B., Georgantas, N., Issarny, V.: Cocoa: Conversation-based service composition in pervasive computing environments with qos support. Journal of Systems and Software 80(12), 1941–1955 (2007)

41. Mokhtar, S.B., Kaul, A., Georgantas, N., Issarny, V.: Efficient semantic service discovery in pervasive computing environments. In: Middleware (2006)
42. Nitto, E.D., Rosenblum, D.S.: Exploiting adls to specify architectural styles induced by middleware infrastructures. In: Proc. of International Conference on Software Engineering, ICSE (1999)
43. Paolucci, M., Kawamura, T., Payne, T.R., Sycara, K.P.: Semantic matching of web services capabilities. In: International Semantic Web Conference, ISWC (2002)
44. Raskin, R.G., Pan, M.J.: Knowledge representation in the semantic web for earth and environmental terminology (SWEET). Computers & Geosciences 31(9), 1119–1125 (2005)
45. Shadbolt, N., Berners-Lee, T., Hall, W.: The semantic web revisited. IEEE Intelligent Systems 21(3), 96–101 (2006)
46. Shaw, M.: Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. In: ICSE Workshop on Studies of Software Design. pp. 17–32 (1993)
47. Spitznagel, B., Garlan, D.: A compositional formalization of connector wrappers. In: International Conference on Software Engineering, ICSE (2003)
48. Tanenbaum, A., Van Steen, M.: Distributed systems: principles and paradigms - Second Edition. Prentice Hall (2006)
49. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software architecture : foundations, theory, and practice. Hoboken (N.J.) : Wiley (2009)
50. Uckelmann, D., Harrison, M., Michahelles, F.: Architecting the internet of things. Springer (2011)
51. Vaculín, R., Neruda, R., Sycara, K.P.: The process mediation framework for semantic web services. International Journal of Agent-Oriented Software Engineering, IJAOSE 3(1), 27–58 (2009)
52. Wiederhold, G.: Interoperation, mediation, and ontologies. In: Proc. of the Fifth International Symposium on Generation Computer Systems Workshop on Heterogeneous Cooperative Knowledge-Bases. pp. 33–48. Citeseer (1994)
53. Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM Trans. Program. Lang. Syst. (1997)